Automatic Deployment of Transcoding Components for Ubiquitous, Network-Aware Access to Internet Services

Xiaodong Fu, Weisong Shi, and Vijay Karamcheti
Department of Computer Science
Courant Institute of Mathematical Sciences
New York University
{*xiaodong,weisong,vijayk*}*@cs.nyu.edu*

Abstract

Advances in wireless communication together with the growing number of mobile end devices hold the potential of ubiquitous access to sophisticated internet services; however, such access must cope with an inherent mismatch between the low-bandwidth, limited-resource characteristics of mobile devices and the high-bandwidth expectations of many content-rich services. One promising way of bridging this gap is by deploying application-specific components on the path between the device and service, which perform operations such as protocol conversion and content transcoding. Although several researchers have proposed infrastructures allowing such deployment, most rely on static, hand-tuned deployment strategies restricting their applicability in dynamic situations.

In this paper, we present an automatic approach for the dynamic deployment of such transcoding components, which can additionally be dynamically reconfigured as required. Our approach relies on three components: (a) a high-level *integrated type-based specification of components and network resources*, essential for "late binding" components to paths; (b) an *automatic path creation strategy* that selects and maps components so as to optimize a global metric; and (c) *system support for low-overhead path reconfiguration*, consisting of both restrictions on component interfaces and protocols satisfying application semantic continuity requirements. We comprehensively evaluate the effectiveness of our approach over a range of network and end-device characteristics using both a web-access scenario where client performance is for reduced access time, and a streaming scenario where client preference is for increased throughput. Our results verify that (1) automatic path creation and reconfiguration is achievable and does in fact yield substantial performance benefits; and (2) that despite their flexibility, both path creation and reconfiguration can be supported with low run-time overhead.

1 Introduction

The role of the Internet has undergone a transition from simply being a data repository to one providing access to a plethora of sophisticated network-accessible services such as e-mail, banking, on-line shopping and entertainment. Additionally, these services are increasingly being accessed by mobile consumers using end devices such as PDAs, Pocket/Handheld PCs, cellular phones and two-way pagers that connect to the internet using a variety of wireless networking options ranging from Bluetooth [7] to Wireless 3G [18]. The combination of these two trends holds out the possibility of providing a user with seamless, ubiquitous access to a service irrespective of the user's end device and location. Although compelling, achieving this goal requires coping with the inherent mismatch between the low-bandwidth, limited resource characteristics of wireless mobile devices and the high-bandwidth expectations of many content-rich services.

This mismatch is particularly troublesome given the existing view, which hides network characteristics from the application and treats services as standalone entities. A typical mobile user is connected to

Report Documentation Page	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE **2001**	2. REPORT TYPE	3. DATES COVERED **-**

4. TITLE AND SUBTITLE **Automatic Deployment of Transcoding Components for Ubiquitous, Network-Aware Access to Internet Services**	5a. CONTRACT NUMBER 5b. GRANT NUMBER 5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER 5e. TASK NUMBER 5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Defense Advanced Research Projects Agency,3701 North Fairfax Drive,Arlington,VA,22203-1714**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S) 11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES
The original document contains color images.

14. ABSTRACT
see report

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**22**	

the internet through multiple types of links with very different bandwidth, delay, and error characteristics ranging from a high-bandwidth WAN link between the central server and an edge server, a broadband link between the edge server and the wireless network the user is on, finally to the wireless link connecting the user's device to this network. These differences, combined with the fact that the nodes along the path can also possess very different capabilities (most true of the end device) produce unsatisfactory performance for network-oblivious applications.

Current day applications and services cope with the above problems essentially by providing differentiated service for different networks/end-devices. For example, most popular news, e-mail, and stock trading services today present a different front-end for mobile users. Although adequate in some scenarios, this approach suffers from the limitation that mobile users are classified into a small number of classes and may not receive performance commensurate with the capabilities of the device or network they are using. More importantly, such an approach cannot adequately cope with dynamically changing environments where there is a big variation in available bandwidth (e.g., a user on a wireless LAN who is at different distances from an access point). More promising are programmable infrastructures recently proposed by several researchers [5, 6, 22], which allow the dynamic injection of application-specific components in the network path; these components cope with device and network mismatches by handling activities such as protocol conversion and content transcoding at sites best suited for them.

Although several such infrastructures have been proposed, they have not seen widespread use because of concerns about their deployability, performance, and scalability. Chief among these concerns, and the focus of this paper, is the question of automatically determining which components must be present along a path at any time to cope with (possibly dynamically changing) network and device characteristics and differing user contexts. This problem has received a great deal of attention recently [6, 16, 15, 9, 12], with suggested solutions broadly falling into two categories: those that lookup a database of precomputed paths for the best match to a given network situation [16, 12], and those that dynamically search the space of all possible data paths to find either a reasonable or optimal path [6, 16, 15, 9]. The first set of solutions offer reduced path set up times at the cost of optimality, flexibility, and adaptability. The second set of solutions has so far focused mostly on ensuring that the components are *functionally compatible* in that they are able to take data produced by the source and convert it to a form consumable by the end device. However, such solutions have tended to neglect dynamic and heterogeneous network characteristics, with the consequence that the performance of the generated path falls well short of satisfying user expectations.

In this paper, we address these shortcomings by describing an automatic approach for deploying network-aware access paths that can additionally be dynamically configured as required. Our approach relies on three components:

- A high-level *type-based specification of components and network resources*, which both reduces the burden on the component developer (who can focus on functionality without worrying about how a component is going to be used along a path) and more importantly enables "late binding" of components to paths, essential for flexibility. While some other researchers have proposed similar formulations, what distinguishes our approach is that the same type-based framework is also used to capture the characteristics of network resources. For example, *network links are represented simply as entities that transform the type of data passing across them*.
- An *automatic path creation strategy*, which finds a type-compatible component sequence that transforms the data type produced at the service into a type that can be consumed by the client device, and additionally respects network resource constraints. Our strategy relies on a dynamic programming-based polynomial-time algorithm that optimizes a global metric (available client throughput or response time) by determining both which components constitute the sequence and how they are mapped to underlying network resources.
- *System support for low-overhead path reconfiguration*, which includes restrictions on component in-

terfaces and efficient protocols that leverage these restrictions to support three different reconfiguration semantics: no continuity, continuity at the level of *semantic segments*, and full continuity.

We have implemented the three components of our approach in the CANS infrastructure, a Java-based application-level infrastructure for constructing network-aware access paths. We comprehensively evaluate the effectiveness of our approach, in terms of both the performance of the constructed paths as well as the overheads associated with path creation and reconfiguration, under multiple network and end-device characteristics reflecting typical mobile use situations for both a web access scenario where user preference is for reduced access time, and a streaming scenario where client preference is for increased throughput. Our results verify that (1) automatic path creation and reconfiguration is achievable and does in fact yield substantial performance benefits; and (2) that despite their flexibility, both path creation and reconfiguration can be supported with low run-time overhead.

The rest of this paper is organized as follows. Section 2 presents an overview of the CANS infrastructure. The framework of component and link types is described in Section 3 and Section 4 presents our planning algorithm that builds on this framework. 5 describes the system support for path reconfiguration. Section 6 evaluates this algorithm with the help of two case studies. We discuss related work in Section 7 and conclude in Section 8.

2 Background: CANS Infrastructure

Composable **A**daptive **N**etwork **S**ervices (CANS) is an application-level infrastructure for injecting application-specific components into the network path between a client and a service. Traditionally, the functionality of a data path is restricted to transmitting data between the end points. The CANS infrastructure extends this notion to enable end services, client applications, or some other entity to dynamically inject application-specific components into the network; these components customize the data path with respect to the characteristics of the underlying physical network links and properties of the end device as well as dynamically adapt to any changes in these characteristics (see Figure 1(a)).

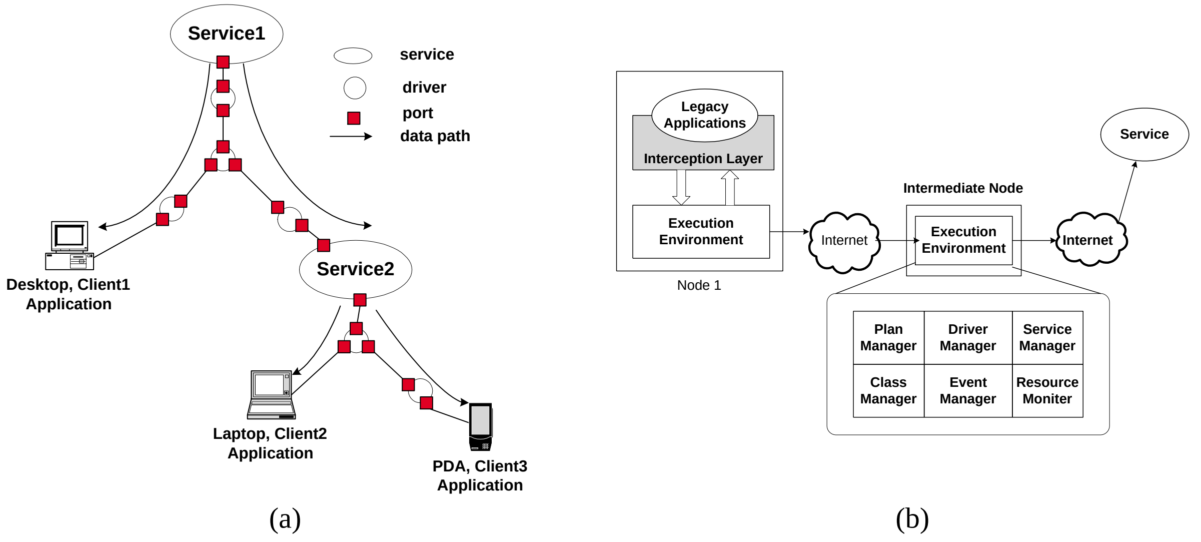


Figure 1: (a) Basic organization of CANS, (b) Interaction between legacy applications and CANS architecture.

The CANS network view consists of *applications*, stateful *services*, and *data paths* between them built up from mobile soft-state objects called *drivers*. Drivers implement a restricted interface (for example, reading and writing data from an implementation-neutral data port interface), which permits efficient composition and semantics-preserving adaptation. Both services and data paths can be dynamically created and reconfigured: a planning and event propagation facility enables distributed adaptation, and a flexible type-based composition model dictates how new services and drivers are integrated with existing ones. The type-based compatibility framework is described in additional detail in Section 3. The CANS network is realized by partitioning the services and data paths onto physical hosts, connected using existing communication mechanisms. The CANS Execution Environment serves as the basic run-time environment on these hosts and consists of several modules as shown in Figure 1(b). The plan manager implements the planning algorithm described in Section 4, and together with the other modules, supports dynamic creation, migration, and adaptation of drivers and services. CANS has been implemented on Windows 2000 clients and Java/RMI-capable intermediate hosts.

CANS distinguishes itself from other infrastructures permitting component injection by (a) supporting legacy applications and services, and (b) enabling configuration and distributed adaptation of injected components in response to system conditions.

Legacy applications interface with the injected components using an *interception layer* (see Figure 1(b)) that transparently virtualizes the network bindings of the application, in our case TCP sockets. Logically, data to and from the application using a particular socket is sent via multiple CANS components, while retaining the illusion from the application's perspective of an end-to-end TCP connection. Legacy services are just as easily integrated; a *delegate object* controls and represents a service in its interactions with the CANS infrastructure.

CANS configures data paths that are customized to network characteristics and user preferences (e.g., minimum response time or maximum throughput) by selecting and deploying an appropriate sequence of components. CANS also supports incremental reconfiguration of data paths in response to dynamic changes in system characteristics. Such reconfiguration, which is accomplished without violating the semantics of the data path, leverages two restrictions placed on driver functionality. The first restriction, of having drivers consume and produce data in application-specific units called *semantic segments*, permits the system to keep track of which data units at the input to a data path segment influence the data units arriving at the output of the segment. The second restriction, requiring drivers to contain only *soft state*, permits the system to reconfigure a data path simply by buffering and retransmitting appropriate semantic segments via freshly created drivers. Additional description of the reconfiguration algorithm can be found in Section 4.

The overall CANS architecture has been presented in a prior publication [5]. In this paper, we describe in additional detail the type framework underlying component composition and the planning algorithm for setting up and reconfiguring data paths.

3 Component Selection as Type Compatibility

We formulate the problem of determining which application-specific components must lie on the data path connecting an end device to the service to best cope with any mismatches in network and device characteristics as a *type compatibility* problem. Central to this formulation is the notion that all data flowing along a data path is *typed*, and that this type is affected both by components along the data path as well as network links making up the route. Component selection then becomes the problem of finding an appropriate selection of components that can be mapped to physical resources in a fashion that permits the data type produced at the service to be transformed into a data type that can be consumed by a client application running at the end device, taking into consideration the type changes induced because of network links along the route. Additional criteria driving the selection include satisfying constraints imposed by node and link capacities and optimizing some overall path metric such as response time or throughput.

In the rest of this section, we describe in turn the type-based representation of components and links.

3.1 Representing Component Properties

The composability of CANS components (both drivers and services) is decided by compatibility of the type information associated with the input and output ports being connected. The types used in CANS integrate two closely related concepts: *data types* and *stream types*.

CANS data types are the basic unit of type information, represented by a type object that in addition to a unique type name can contain arbitrary attributes and operations for checking type compatibility. Traditional mechanisms such as type hierarchies can still be used to organize data types; however, our scheme permits flexible type compatibility relationships not easily expressibly just by matching type names. For instance, it is possible to define a CANS type for MPEG data, which contains attributes for defining the frame size. An `MPEG` type can be defined compatible with another `MPEG` type as long as the former's frame size is smaller than the latter's, naturally capturing the behavior that a lower resolution MPEG stream can be played on a client platform capable of displaying a higher resolution stream.

CANS stream types capture the aggregate effect of multiple CANS drivers operating upon a typed data stream. Stream types are constructed at run time, and representable as a *stack* of data types. Operations allowed on stream types include *push, pop, peek,* and *clone*, which have the standard meanings.

Each CANS component with m input ports and n output ports defines a function, which maps its input stream types into output stream types:

$$f(T_{in_1}, T_{in_2}, ..., T_{in_m}) \rightarrow (T_{out_1}, T_{out_2}, ..., T_{out_n})$$

where T_{in_i} is the required stream type set for the ith input port, and T_{out_j} is the resulting stream type produced on the jth output port. The type compatibility between an input and an output port, which determines whether two components can be connected, is determined by checking the top of the output port's stream type against the required data type of the input port. Stream type information flows downstream automatically when two ports get connected at run time.

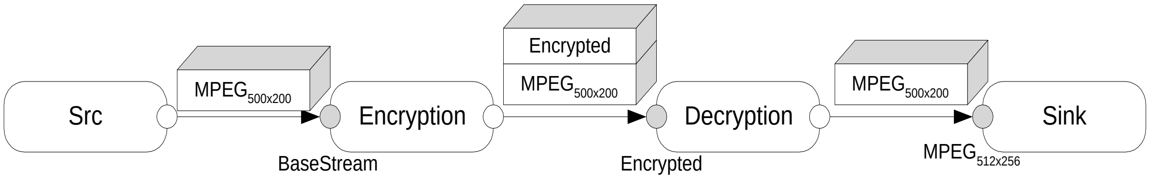


Figure 2: A simple example of type compatibility.

Figure 2 shows an example of the type compatibility scheme. The source produces MPEG data at resolution 500×200, which needs to be supplied to the sink that can consume MPEG data at resolution 512×256 after going through two components that respectively encrypt and decrypt the data. The figure shows the data types on each of the ports as well as the stream types on the connections. To consider an example, the *Encryption* driver accepts data type `BaseStream` and pushes an `Encrypted` type object onto the incoming stream type. The output port of *Src* is compatible with the input port of *Encryption* because the `MPEG` type object extends the `BaseStream` type. Similarly, the output port of *Decryption*, whose affect is to pop the `Encrypted` type from its incoming stream type, is compatible with the input port of *Sink* because of a type-specific compatibility operator for the `MPEG` type that looks at the resolution attributes.

Figure 2 also highlights the composition advantages of representing stream types as a stack of data types. If components were just modeled as consuming data of a particular type and producing data of another type, it would be difficult to express the behavior of the *Encryption* and *Decryption* drivers in a way that permits their use for a variety of generic stream types *without* losing information about the original stream type at the output of the *Decryption* driver. Thus, determining whether the *Decryption* driver's output port is compatible with the input port on *Sink* would require examining the entire data path. In contrast, our stream type representation permits local decision making, a prerequisite for run-time adaptation via dynamic component composition.

3.2 Representing Link Properties

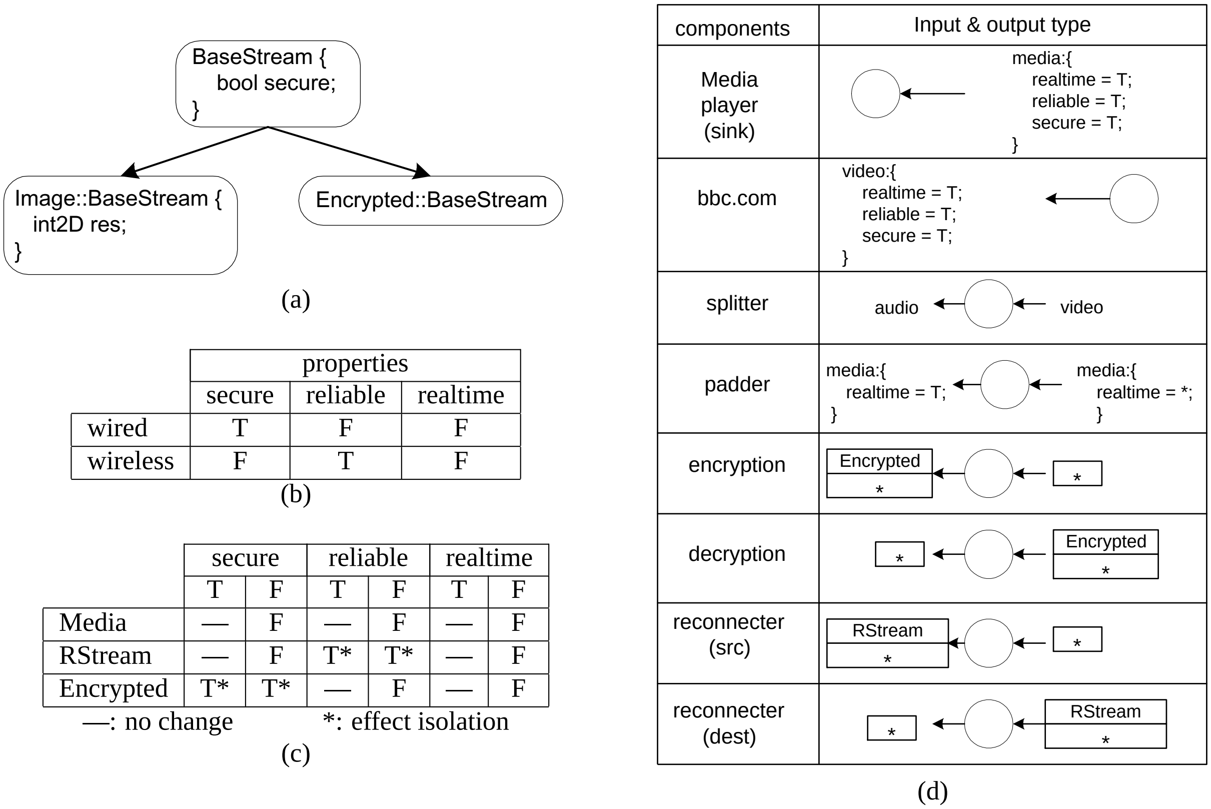
The properties of links making up the route between a service and a client application very closely impact which components must be selected in order to satisfy user preferences. For example, if the route between the service and the user includes an insecure link, the data needs to be encrypted (i.e., pass through an encryption component) prior to crossing this link if the path has to satisfy user preferences of security. Similarly, if a continuous media stream need be transmitted across links where it is not possible to bound jitter, there needs to be a component downstream of the link that can reinstate the real-time property of the stream. Despite recognition of this close coupling, prior research has usually modeled links in an ad hoc fashion inserting components required because of link properties as a separate pass after type-compatibility based selection. While this approach works, it compromises on optimality because of poor or redundant placement of these required components.

In contrast, our approach unifies both type compatibility and link properties in selecting which components need to be present. The basic idea is to represent link requirements implicitly by modeling how links effect the types of data that go across them. To capture the effect of link properties on data types, we introduce the notion of an *augmented type*: each data type is extended with a set of link properties that can take values from a fixed set such as security, reliability, and timeliness. Network links are modeled in terms of the same properties and have the effect of modifying, in a type-specific fashion, values of the corresponding properties associated with different data types. To consider an example, consider transmission of HTML data over an insecure link. Our type framework captures this as follows: the data type produced at the source is represented by `HTML`(*secure*=`true`), the network link is represented by the property *secure*=`false`, and the effect of the link property *secure* on the `HTML` data type by the rule that the augmented type `HTML`(*secure*=`true`) is modified to `HTML`(*secure*=`false`) upon crossing a link with the property *secure*=`false`.

This base scheme is extended to stream types by introducing the notion of *isolation*. Stated informally, some data types have the capability to isolate others below them in the stream's type stack from having their properties be affected by a link. For example, the `Encrypted` type isolates the *secure* property of types that it "wraps", i.e., encrypted data still remains secure after crossing insecure links, irrespective of what specific type(s) the data corresponds to.

3.3 Example: Access to Streaming Media

Given the type framework described above, component selection is driven by four pieces of information: data type definitions, component properties described in terms of input and output types, links modeled in terms of their link properties, and rules governing how data types are modified by links. We describe these components for an example scenario where a mobile user who is connected to the internet with both wired and wireless connectivity options (e.g., a laptop capable of both connected and mobile operation) accesses a media stream from an internet-based server. The usage scenario consists of the user starting off using the wired connection but switching across to the wireless connection in the middle. The user preference is to receive a continuous real-time stream which is guaranteed transmitted in a secure fashion inspite of the connection transition and limited security of the wireless link. These user preferences are handled by using the CANS infrastructure to automatically deploy components, which insulate the application from the switch in connections and offer required security guarantees. The type components of this example are described below:



	properties		
	secure	reliable	realtime
wired	T	F	F
wireless	F	T	F

(b)

	secure		reliable		realtime	
	T	F	T	F	T	F
Media	—	F	—	F	—	F
RStream	—	F	T*	T*	—	F
Encrypted	T*	T*	—	F	—	F

—: no change *: effect isolation

(c)

Figure 3: Types in the streaming media example: (a) data type definitions; (b) link properties; (c) effect of link properties on augmented types; and (d) input and output types of components.

Figure 3(a) shows the data type definitions. `BaseStream` is the basic stream type with three boolean link properties, *reliable*, *secure* and *realtime*. `RStream`, `Media`, and `Encrypted` extend the `BaseStream` type, representing reliable, media, and encrypted streams respectively. `Video` and `Audio` are two subtypes of the `Media` type. The non-media types are produced and consumed by auxillary components described below.

Figure 3(b) shows the properties of the wired and wireless links. The wired link is modeled with *reliable* and *realtime* properties set to `false` to capture the fact that it can get disconnected during the access. Figure 3(c) shows how these link properties affect different types, with "effect isolation" referring to a type isolating the effect of a link property for data type instances below it in the type stack. For example, the security property of the `Encrypted` type is unaffected when data of such type traverses an insecure link. Moreover, the type isolates this effect on any of the wrapped types.

Figure 3(d) lists the input/output types of six components, along with the types produced by the source and that required by the sink. The *splitter* component splits a video+audio stream into an audio-only stream while the *padder* "fills in" legal media frames whenever its input stream stops. The behavior of the padder component is represented by the transformation of its input type (`RStream` with an arbitrary value associated with the *realtime* property) into its output type (`RStream` with *realtime*=`true`). The other components cooperate to handle encryption (*encryption* and *decryption*) and reliable transmission (*reconnecter(src)* and *reconnecter(dest)*) respectively. These components are required to overcome unfavorable link properties associated with the wired and wireless links, specifically the *secure* and *reliable* properties.

Thus, legal type-compatible component sequences for transmitting the media stream to the client include:

- (using the wired link) The *reconnecter(src)—reconnecter(dest)—padder* sequence, when link capacities are sufficient for transmission of the original video+audio stream to the client. When link capacities are not sufficient, the sequence would need to include the *splitter* component at a location determined by the bottleneck link.
- (using the wireless link) The *encrypter—reconnecter(src)—reconnecter(dst)—decrypter—padder* sequence, when link capacities are sufficient to transmit video+audio to the client. As before, when not enough capacity is present, the sequence would need to include the *splitter* component.

Section 6 shows, for a web page access scenario, how given such high-level type specifications, our planning algorithm automatically chooses an optimal sequence and maps it to underlying resources.

4 Selection and Mapping of Components

The goal of the CANS planning algorithm is to select and map a type-compatible set of components to the underlying network resources in response to a request from a client application to connect to an end service. The planning procedure, implemented by the plan manager component of the execution environment, consists of two steps: *route selection* where a graph of nodes and links is selected for deploying the plan, and *component selection* where appropriate components are selected and mapped to the selected route.

Route selection can be viewed as the shortest path problem in the node graph, which takes into consideration bandwidth on links between nodes in different domains and the relative loads on nodes within the same domain. Given the large amount of literature available on similar problems, we will not discuss this further.

The component selection process, described in additional detail below, takes as input the augmented type at the data source, the augmented type required at the sink, and the selected route (whose links may transform augmented types as described earlier). In this paper, we restrict our attention to single input, single output components; i.e., all selected plans consist of a sequence of components. We use a dynamic programming algorithm to simultaneously select a component and map it to the route in a fashion that optimizes overall throughput. For clarity of presentation, we first describe a base version of the algorithm where only simple (non-stacked) data types are present and links *do not* affect the type of data crossing them, and then discuss how this base algorithm can be extended to handle the more general case of stream types and link properties. We conclude this section by describing the path reconfiguration process.

4.1 Base Algorithm

To describe the dynamic programming algorithm, we first need to introduce some terminology.

A **driver component** d is modeled in terms of its *computation load factor*, $\mathrm{load}(d)$, and its *bandwidth impact factor*, $\mathrm{bwf}(d)$. $\mathrm{load}(d)$ captures the per-input byte cost of running the component, while $\mathrm{bwf}(d)$ reflects the average ratio of input and output bandwidths. For example, a compression component that reduces stream bandwidth by a factor of two has a $\mathrm{bwf} = 0.5$.

A **data path**, $D = \{d_1, \ldots, d_n\}$, is a sequence of type-compatible components, as defined in Section 3. A **type graph** formalizes this notion: vertices in the graph represent types, and edges represent components that can transform the type of one vertex to the type of the other. There might be multiple edges between two vertices in the type graph; the degree of a vertex does not exceed the number of components in the system.

A **route**, $R = \{n_1, n_2, \ldots, n_p\}$, is a sequence of nodes obtained using the route selection algorithm. $R(n_i, n_j)$ refers to the subsequence starting at node n_i and ending at node n_j. Each node n_i is modeled in terms of its *computation capacity*, $\mathrm{comp}(n_i)$, which represents the number of operations that the node can perform per unit time. A link between two nodes, l_{ij}, is modeled in terms of its bandwidth, $\mathrm{bw}(l_{ij})$.

A **mapping**, $M : D \rightarrow R$, associates components on data path D with nodes in route R. We are only interested in mappings that satisfy the following restriction: $M(d_i) = n_u, M(d_{i+1}) = n_q \Rightarrow u \leq q$; i.e., components are mapped to nodes according to sequence order. This is a reasonable assumption for data paths crossing multiple networking domains.

The component selection process takes as its input a route R, a source data type t_s, a destination data type t_d, and attempts to find a data path D that transforms t_s to t_d and can be mapped to R to yield maximum throughput.

The problem as stated above is NP-hard. To make the problem tractable, we view the computation capacity as partitionable into a *discrete* number of load intervals; i.e., capacity is allocated to components only at interval granularity. Not only is this assumption practical, but it also allows us to define, for a route R, the notion of an *available computation resource vector*, $\vec{A}(R) = (r_1, r_2, \ldots, r_p)$, where r_i reflects the available load intervals on node n_i (normalized to the interval [0,1]). For this algorithm, we are interested only in a subset of all possible vectors that have the pattern $\{1, \ldots, 1, r_i, 0, \ldots, 0\}$. Informally, these legal vectors represent situations of a left-to-right allocation strategy, where only a single node in the route is being considered for mapping a component at a time, all nodes after it is not available . It can be easily verified that the total number of such legal vectors is $p \times L$, where p is the number of nodes and L is the number of the discrete load intervals.

Dynamic Programming Strategy

The algorithm builds up partial optimal solutions, $s[t, \vec{A}, k], \forall t, \vec{A}, k$, where each such solution yields maximum throughput for transforming the source type t_s to an arbitrary intermediate type t, using a data path with k components or fewer and requiring no more resources than $\vec{A}$. The dynamic programming strategy defines how these solutions can be constructed in a bottom up fashion:

- Step 1 solutions simply consist of single-component paths (edges in the type graph) that transform t_s into an arbitrary intermediate type t, and require no more than $\vec{A}$ resources (for each $\vec(A)$ for a given route R).
- To explain how the algorithm works, assume that Step $k-1$ solutions have been constructed. These consist of optimal paths of $k-1$ or fewer components that transform the source type into an intermediate type while using no more than $\vec{A}$ resources (for each $\vec{A}$). The dynamic programming step works as follows.
- To construct a Step k solution for a given type t and resource vector $\vec{A}$, consider all possible intermediate types t' that can be transformed to t; i.e., all those types for which an edge (t', t) is present in the type graph. For each such t', consider all possible mappings of the associated component d on nodes along the route that use no more than $\vec{A}$ resources. For each such mapping that transforms the available resource vector to $\vec{A'}$ (after accounting for $\mathrm{load}(d)$), combine this component with the optimal Step $k-1$ solution $s[t', \vec{A'}, k-1]$. The combined mapping that yields the maximum throughput is deemed the optimal Step k solution.

The throughput achievable for a particular mapping can be computed given the node throughput and link bandwidth properties. The throughput of node n_i itself is decided by the incoming bandwidth, its computation capacity $\mathrm{comp}(n_i$, and the load and bwf properties of components mapped to the node. One additional point needs some clarification: in the above algorithm, we need to know how much resources to set aside for component d before we can combine d with an optimal Step $k-1$ solution. The problem here is that d's resource requirements $\mathrm{load}(d)$ are expressed in terms of per-input byte costs, and are difficult to evaluate without knowing what the input bandwidth is, which itself is only known once the Step $k-1$ solution is selected. Our solution to break this cyclic dependency is to first *guess* the resource requirement of d and then evaluate the throughput for this guess. The guess that yields the maximum throughput is picked to reflect d's resource usage. Note that because of discretized load levels, we only need to make a constant number of guesses at each step.

The algorithm terminates at Step $k_{\max} = p \times n$, where p is the number of nodes and n is the number of components. This follows from the observation that for real components, there is no throughput benefit from mapping multiple components to the same node. The solution $[t_d, \vec{A}_{\max}, k_{\max}]$, if present, yields the optimal selection and mapping of components to transform t_s to t_d along route R. The complexity of this algorithm is $O(p^3 n^3)$.

4.2 Extension 1: Handling Stream Types

Stacked stream types complicate the structure of the type graph, because the latter needs to capture the fact that the same driver component can now be used to bridge among many different input types. For example, a `Zip` compression driver can consume data of any type. One simple way of handling such stream types is to insert nodes into the type graph that explicitly enumerate all possible stack configurations. In the above example, there would be nodes such as `Zip-HTML` and `Zip-MPEG` corresponding to each data type passing through the `Zip` driver. Although this approach would correctly handle stacked types, it is not very practical because the size of the type graph can be exponential in the number of simple data types.

To ensure that the type graph does not become intractably large, we employ two strategies. First, we restrict the type graph to include only those stream types that are reachable from the source data type, and which in turn can reach the destination type required by the client. Second, we rank the primitive data types and introduce the constraint that only types of monotonically increasing ranks can be stacked into a stream type. Such rank ordering not only reduces the size of the type graph, but can also be used to introduce application-specific constraints on how CANS components can be composed. For instance, we can ensure, for any CANS data path requiring both encryption and compression, that encryption always happens after compression by giving the encryption type a higher rank. Similarly, and this is something that our web access case study described in Section 6 exploits, we can capture requirements that say for instance that image resizing (to reduce bandwidth requirements) should only be employed after image quality filtering. In our case study, these two strategies reduce the size of the type graph to just 3 when planning for the data type `mime/text` and 6 when planning for the data type `mime/image`. In contrast, the application drivers repository included about twenty simple types that would have resulted in a substantially larger number of nodes otherwise.

4.3 Extension 2: Dealing with Link Properties

The algorithm as described so far does not consider the possibility of network links affecting stream data types. To cope with the latter, the algorithm needs to incorporate two modifications. First, the type graph is now defined in terms of augmented types, making explicit the differences between streams that have the same data type but different values of link properties such as security. Because both the number of such link properties as well as the set of values associated with each property are expected to be small, such enumeration results in only a small increase in the size of the type graph.

The other modification is to the recursive step in the dynamic programming algorithm described above. In particular, when developing Step k solutions, the optimal Step $k-1$ solution that is combined with the selected one-component partial mapping must take into consideration possible type translations because of an intermediate link. In other words, for a given intermediate type t', we now need to consider all solutions $s[t'', \vec{A'}, k-1]$ where t'' is translated by the link into t'. This modification does not change the overall complexity of the algorithm.

5 System Support for Efficient Path Reconfiguration

Network-aware access paths need to be reconfigured to cope with dynamic changes in user preferences or network resource characteristics. Our approach relies on two kinds of system support to enable low-overhead reconfiguration: (1) appropriate restrictions on component interfaces, and (2) reconfiguration protocols that leverage these restrictions. In this section, we first describe the reconfiguration semantics supported by CANS, and then the required system support.

5.1 Reconfiguration Semantics

The central question about reconfiguration is what can the application assume about data in transit or buffered within components when a portion of the network path is reconfigured. CANS reconfiguration protocols can be customized to provide three levels of semantics:

- **Case 1** semantics provides no guarantees, leaving it up to the application to reconstruct any lost data. Applications involving non-critical data (e.g., news feeds) can exploit in-order delivery guarantees to perform efficient recovery.
- **Case 2** semantics provides the guarantee of delivering complete *semantic segments*, essentially simplifying the task of the application recovery code. Semantic segments represent application-specific notions of a useful granularity of data. For example, in a streaming media application, a semantic segment might correspond to individual frames. Case 2 semantics ensure that a frame is either completely delivered or not delivered at all.
- **Case 3** semantics provide full continuity guarantees with exactly-once semantics, completely isolating the application from the fact that the path has been reconfigured. Note that real-time applications can still detect a break in data availability; we take the view that such applications are best handled by inserting additional application-specific components that provide necessary timeliness guarantees.

5.2 Restrictions on the Driver Interface

To guarantee the above semantics, drivers are required to adhere to a somewhat restricted interface. Specifically,

1. Drivers are written to consume and produce data using a standard typed *data port* interface, called a `DPort`.
2. Drivers are *passive*, moving data from input ports to output ports in a purely demand-driven fashion. Driver activity is triggered when one of its output ports is checked for data, or one of its input ports receives data.
3. Drivers consume and produce data at the granularity of an integral number of application-specific *semantic segments*. These segments are naturally defined based on the application, e.g., an HTML

page or an MPEG frame. This requirement ensures that the CANS infrastructure is made aware of "markers" in the data stream, which it uses to guarantee Case 2 and Case 3 semantics. Note also that this property only refers to the logical view of the driver, and admits physical realizations that transmit data at any convenient granularity as long as segment boundaries are somehow demarcated (e.g., with marker messages).

4. Drivers contain only *soft state*, which can be reconstructed simply by restarting the driver. Stated differently, given a semantically equivalent sequence of input segments, a soft-state driver always produces a semantically equivalent sequence of output segments. For example, a Zip driver that produces compressed data will produce semantically equivalent output (i.e., uncompressed to the same string) if presented with the same input strings.

The first two properties enable dynamic composition and efficient transfer of data segments between multiple drivers that are mapped to the same physical host (e.g., via shared memory). Moreover, they permit driver execution to be orchestrated for optimal performance. For example, a single thread can be employed to execute, in turn, multiple driver operations on a single data segment, achieving nearly the same efficiency as if driver operations were statically combined into a single procedure call. The semantic segments and soft-state properties enable low-overhead path reconfiguration as described in additional detail below, while preserving application semantics.

5.3 Reconfiguration Protocol

Path reconfiguration is triggered by events generated either by the CANS execution environment, which monitors network resource characteristics, or by a component that detects a change in an application-specific quality metric. The reconfiguration process consists of three major steps: (1) generation of a new plan by a distinguished coordinator node; (2) ensuring required semantics prior to freezing data transmission; and (3) deploying the new plan and resuming data transmission. Step 1 uses the planning algorithm described earlier, optionally reusing some of the partial solutions constructed during initial deployment. Step 3 involves a standard two-phase commit like procedure to synchronize reconfiguration activities among multiple nodes along the path. We describe Step 2 in additional detail below.

Step 2 requires slightly different support for the three reconfiguration semantics described earlier. Since activities for cases 1 and 2 are a subset of that for case 3, our description focuses on the latter. The underlying problem is that to maintain semantic continuity and exactly-once semantics, any scheme must take into account the fact that the portion of the data path affected by the reconfiguration can have stream data that has been partially processed: in the internal state of drivers, in transit between execution environments, or data that has been lost due to failures. Note that the soft-state requirement on its own does not provide any guarantees on semantic loss or in-order reception.

Figure 4 shows an example highlighting this problem. To introduce some terminology, we refer to the portion of the data path that needs to be reconfigured because of network changes (failures are an extreme example) as the *reconfigurable portion*, and the components immediately upstream and downstream of this portion with respect to the data path as the *upstream point* and *downstream point* respectively. In the example, driver d_0 is a source of MPEG data, driver d_1 is an MPEG frame duplicator which produces 3 frames for each incoming frame, driver d_2 is an MPEG frame composer which generates one MPEG frame upon receiving four incoming frames from d_1, and d_3 is a renderer of MPEG data. The reconfigurable portion consists of drivers d_1 and d_2. Consider a situation where system conditions change after the upstream point d_0 has output two frames, and the downstream point d_3 has received one frame. At this point, the portion containing d_1 and d_2 cannot be reconfigured because doing so affects semantic continuity. It is incorrect to retransmit either the second segment from d_0 whose effects have been partially observed at d_3, or the third segment, which would result in a loss of continuity at d_3.

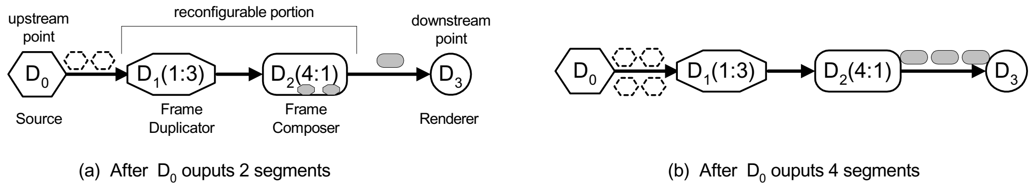


Figure 4: An example of data path reconfiguration using semantics segments.

The reconfiguration protocol leverages the semantic segments and soft state restrictions placed on driver functionality as follows. Intuitively, the first restriction allows us to infer which segments arriving at the downstream point of the reconfigurable portion depend on a specific segment injected at the upstream point and vice-versa, while the second makes it always possible, even if any internal driver state is reset, to recreate the same output segment sequence at the downstream point by just retransmitting selected input segments at the upstream. Our solution exploits these characteristics to provide the required guarantees by just combining buffering and delayed forwarding of semantic segments at the upstream and downstream points respectively with selective retransmission of segments that are incompletely delivered. The correspondence between upstream and downstream segments is completely determined by driver characteristics in the reconfigurable portion; the implementation just needs to track marker messages that demarcate segment boundaries.

This scheme uniformly handles both the situation where drivers continue error-free operation but the data path needs to be reconfigured in response to system conditions, as well as the situation where link or node errors cause partial driver state to be lost; the difference in the two situations is only whether the protocol is executed on demand or always. For the first situation, we defer reconfiguration to the time when the system can guarantee continuity and exactly once semantics for Case 3 (respectively, complete delivery of a semantic segment for Case 2). Upon receiving an event that triggers reconfiguration, the upstream point starts buffering segments while continuing to transmit them, in effect flushing out the contents of intermediate drivers. The downstream point monitors the output segments arriving there, waiting until it *completely receives an output segment from upstream satisfying the property that all subsequent segments correspond only to input segments from upstream point either buffered at the upstream point or not yet transmitted* (or for Case 2 the simpler requirement that all semantic segments that originate from the same input segment are delivered). At this time, the system can be stopped and the reconfigurable portion replaced by a semantically equivalent set of drivers. To restart, the upstream point retransmits starting from the first segment whose corresponding output segment was not delivered.

In our example, reconfiguration works as follows (assuming Case 3 semantics). To start with, the upstream point (d_0) starts buffering every segment it sends out after this time. When the downstream point (d_3) receives a complete upstream segment (in this case this happens when the third segment output by d_2 is received), it raises an event. The plan manager can now freeze d_0, and replace d_1 and d_2 with a compatible driver graph. To restart, d_0 retransmits starting from segment 5. In this case d_3 does not need to discard anything. Error recovery on this portion requires d_0 to buffer its output segments and have the downstream point pass on segments to d_3 only in units of 3 segments at a time.

6 Performance Evaluation

To evaluate our automatic planning and path reconfiguration approach, we measured the performance benefits and run-time overheads achieved by two applications—web access and image streaming—running on top of CANS for a wide range of network resource characteristics. We describe in turn our experimental platform, the performance of the automatic path creation strategy, and the overheads of path reconfiguration.

6.1 Experimental Platform

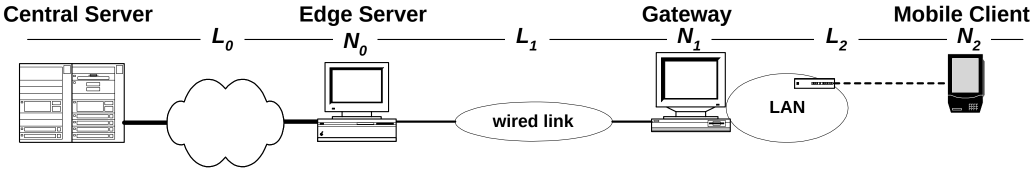


Figure 5: A typical network path between a mobile client and an internet server.

For all of our experiments, we consider a prototypical network path between a mobile client and an internet server as shown in Figure 5. The links involved in such a path include a high-bandwidth WAN link (L_0) between the central server and an edge server (N_0), a broadband link (L_1) between the edge server and the gateway (N_1) to the LAN the user might be on, and finally the wireless link (L_2) connecting the user's device (N_2) to this network.

The **web access application** consists of a browser client and transcoding components that reduce download times under low-bandwidth network conditions (say in a mobile access scenario) by dynamically compressing text and/or degrading image quality. Previous research has shown that such an approach is effective [4, 13]. In this paper, we focus on how, for a range of network resource conditions, to automatically select and map an appropriate set of components so as to minimize download time, taking as input only a high-level specification of the transcoding components and a description of the target network. The **image streaming application** consists of a simple downloadable applet that sets up a connection with a server, receiving and displaying images periodically pushed by the latter. This application is representative of news feeds and tickers on many financial web pages. For our application, we require that images are available at the client within a certain time deadline, and that the transmission is secure.

Component	*Input/Output Types*	*Load* (ops/byte)	*Bandwidth Factor*
ImageFilter	**F**: Image ⟶ Image	1.64×10^{-6}	3.92
ImageResizer	**R**: Image⟶Image	8.335×10^{-6}	3.92
Zip	**Z**: * ⟶ ZipType/*	1.3×10^{-7}	3.15
Unzip	**U**: ZipType/*⟶*	1.2×10^{-7}	0.32
Demultiplexer	**D**: MIME ⟶ Image,Text	negligible	1.0
Multiplexer	**M**: Image,Text ⟶ MIME	negligible	1.0
Encrypter	**E**: * ⟶ Encrypted/*	4.35×10^{-6}	1.0
Decrypter	**D**: Encrypted/* ⟶ *	4.35×10^{-6}	1.0

Table 1: Characteristics of components employed in the web access and image streaming applications.

Table 1 lists the characteristics of components used in the two applications. The *ImageFilter* and *ImageResizer* components degrade image quality and the *Zip* and *Unzip* components work together to compress text pages as required. *Demultiplexer* and *Multiplexer* enable different CANS paths for text and images, and the *Encrypter* and *Decrypter* components help secure the path. The load values in Table 1 are normalized with respect to a Pentium III 1 GHz machine, which has a computing power of 1ops/second. The load and bandwidth factor values were obtained by profiling component execution on representative data inputs: a web page containing 14 KB text and six 24 KB JPEG images for the first application, and a 24 KB JPEG image for the second. All experiments used the same data inputs that the components were profiled on. This is a simplifying assumption, but reasonable given our primary focus was evaluating whether our approach could effectively adapt to multiple network conditions. Evaluating the effectiveness of the approach when component characteristics may be imprecise is a topic deferred to future research.

6.2 Performance of Automatic Component Selection and Mapping

Platform	*Edge Server* (N_0)	L_1	*Gateway* (N_1)	L_2	*Client* (N_2)	*Plan*
1	Medium	Ethernet	High	19.2 Kbps	Cell Phone	A
2	Medium	Ethernet	High	19.2 Kbps	Pocket PC	A
3[*]	High	Fast Ethernet	Medium	57.6 Kbps	Laptop	B
4[*]	High	Fast Ethernet	Medium	115.2 Kbps	Laptop	B
5	Medium	Ethernet	High	384 Kbps	Pocket PC	A
6[*]	High	Fast Ethernet	Medium	576 Kbps	Laptop	B
7[*]	Medium	Fast Ethernet	High	1 Mbps	Laptop	C
8	Medium	Ethernet	High	3.84 Mbps	Pocket PC	D
9	Medium	Ethernet	High	3.84 Mbps	Laptop	D
10	Medium	DSL	High	3.84 Mbps	Laptop	B
11	Medium	DSL	Firewall	3.84 Mbps	Laptop	B
12[*]	Medium	Fast Ethernet	High	5.5 Mbps	Laptop	E

Relative computation power of different node types (normalized to a 1 GHz Pentium III node):
High = **1.0**, Medium = **0.5**, Laptop = **0.5**, Firewall = **0.25**, Pocket PC = **0.1**, Cell Phone = **0.05**
Link bandwidths:
Fast Ethernet = **100 Mbps**, Ethernet = **10 Mbps**, DSL = **384 Kbps**

[*] Experiment conducted on real (as opposed to "sandboxed") hardware.

Table 2: Twelve configurations representing different mobile network connectivity scenarios identifying the CANS plan automatically generated in each case.

To model network conditions likely to be encountered along a mobile access path, we defined twelve different configurations listed in Table 2, representing different network connectivity options and different node capacities.[1] These configurations are grouped into three categories, based on whether the mobile link L_2 exhibits cellular, infrared, or wireless LAN-like characteristics. Four of the configurations correspond to real hardware setups (tagged with a *), the remainder were emulated using "sandboxing" techniques that constrain CPU, memory, and network resources available to an application [3]. As before, the computation power of different nodes is normalized to a 1 GHz Pentium III node.

Table 2 also identifies, for each platform configuration, the plan automatically generated by CANS for the web access application, which are shown in Figure 6. For instance, CANS generates Plan C for platform configuration 7: the placement of the ImageFilter and Zip components on the gateway machine permit adaptation to the low bandwidth link between the gateway and the mobile client. Contrast this plan with plan A (for configurations 1 and 2) where the gateway node now contains both an ImageFilter and an ImageResizer component. The latter is required because the bandwidth reduction due to just the ImageFilter component is insufficient to cross the 19.2 Kbps link.

Figure 7 shows the performance advantages of the automatically generated plans when compared to the response times incurred for direct interaction between the mobile client and the server (denoted Direct in

[1] The bandwidth between the central server and edge server available to a single client is assumed to be 10 Mbps.

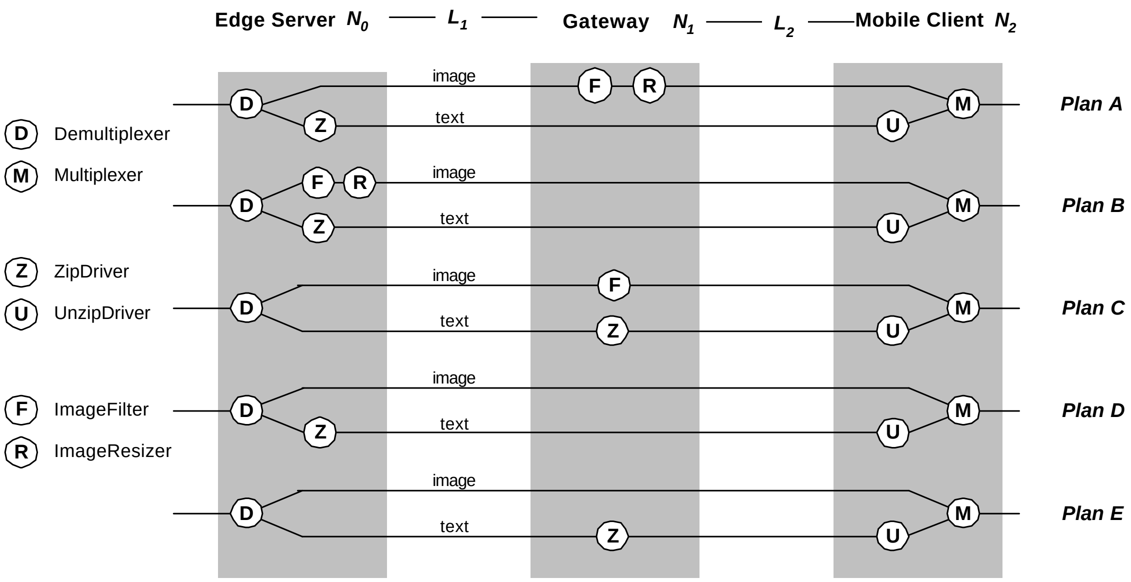


Figure 6: Component placement for the five automatically generated plans.

the figure). The bars in Figure 7 are normalized with respect to the best response time achieved on each platform (so lower is better). In all twelve configurations, the generated plans improve the response time metric, by up to a factor of seven. Note that the lower response times come at the cost of degraded image quality, but this is to be expected. The point here is that our approach *automates* the decisions of when such degradation is necessary. Figure 7 also shows that different platforms require a different "optimal" plan, stressing the importance of automating the component selection and mapping procedure. In each case, the CANS-generated plan is the one that yields the best performance, also improving performance by up to a factor of seven over the worst-performing transcoding path.

It is interesting to note that CANS achieves substantial performance improvements despite run-time overheads on the critical path. To understand whether other applications with different component characteristics would yield similar improvements, we profiled our implementation to construct a timeline of the operations involved in processing a client request for the web page. Figure 8 shows the overall timeline for plan B running on platform configuration 10, and breaks down portions of this timeline into individual operations performed by the CANS execution environment and the components themselves for processing a single text and image packet. The original client request results in the downloading of the text portion of the page, and is followed by requests for each of the six contained images. A text request is received by the edge server N_1, which forwards it to the central server and waits for the latter to respond. Text responses comprise several packets, each of which passes through the Demultiplexer and Zip drivers on the edge server, and the Unzip and Multiplexer drivers on the client before being delivered to the browser application. Similarly a response to an image request comprise multiple packets, each of which flow through the Demultiplexer, ImageFilter, and ImageResizer drivers on the edge server and the Multiplexer on the client before being delivered to the application.

The timeline shows that for this case study, CANS overheads are negligible and dominated by the round-trip between the edge server and the central server (0.2 seconds on the text path and 0.16 seconds on the image path). Even if this were not the case, CANS overheads (shown hatched in the figure) for retrieving data from the network and supplying it to each driver in turn are small for all but very fine-grained components (the Demultiplexer and Multiplexer). For the components used in this study, CANS incurs an average cost of about 25μs per driver invocation, and we expect these overheads to improve significantly as the system is

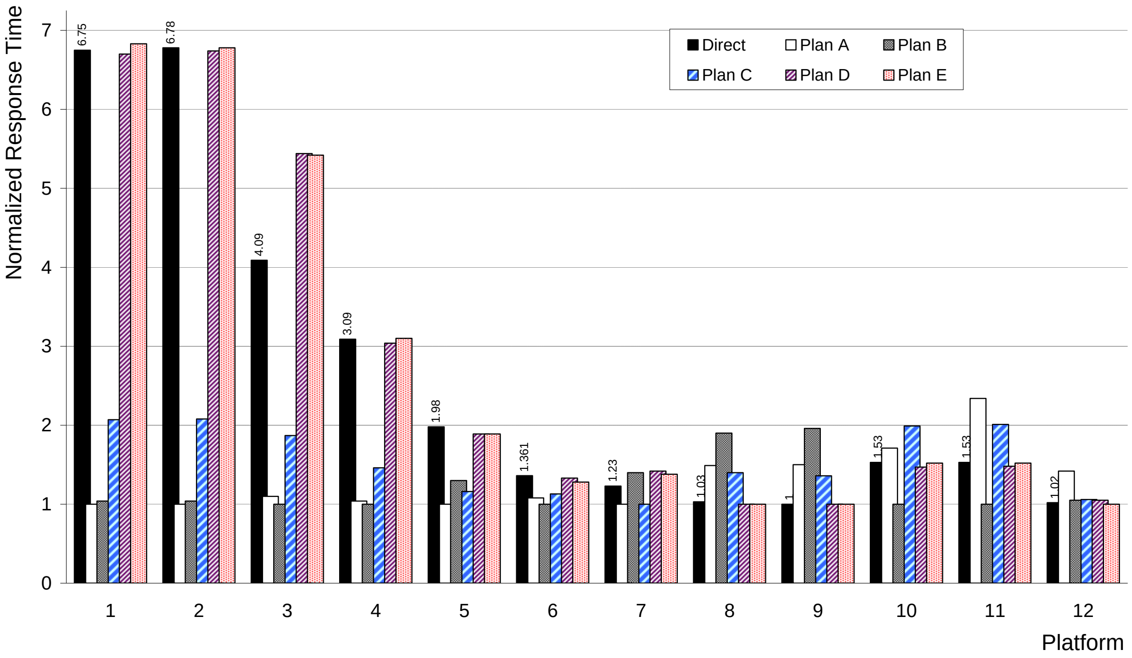


Figure 7: Response times achieved by different plans for each of the twelve platform configurations compared to that achieved by direct interaction. All times are normalized to the best performing plan for each configuration.

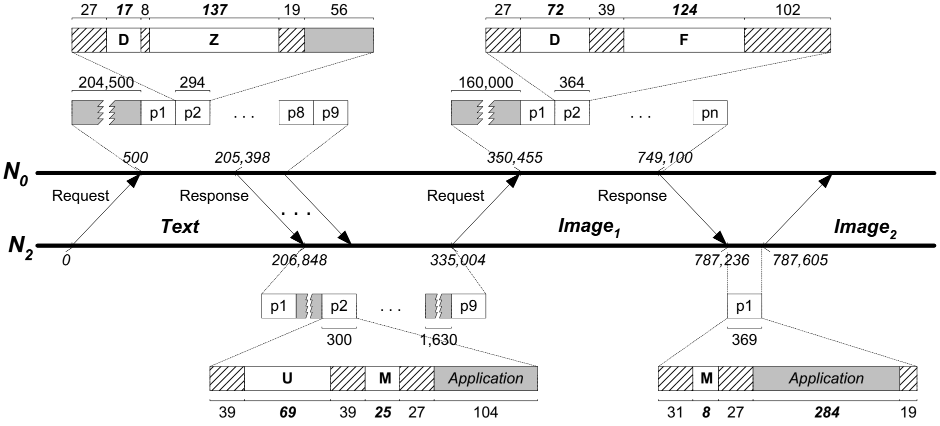
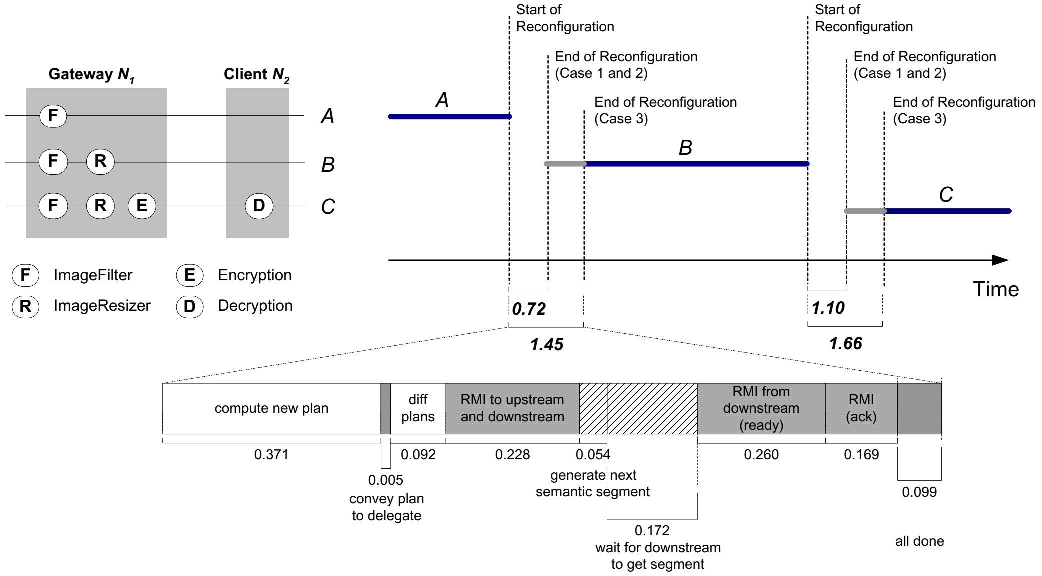


Figure 8: Timeline of requests and responses for plan B running on platform configuration 10 (all times are microseconds). The blocks marked **D**, **M**, **Z**, **U**, and **F** correspond to the executions of the respective components. Communication overheads, including wait times, are shown using gray, whereas CANS overheads are shown using hatched blocks. *Application* refers to the overhead of communicating the data to the client application.

tuned for performance.

6.3 Performance of Data Path Reconfiguration

To evaluate the effectiveness of our path reconfiguration approach, we ran the image streaming application under dynamically changing network conditions, letting the CANS infrastructure automatically generate and reconfigure its access path. For this application, the three levels of reconfiguration semantics correspond to no guarantees about continuity (Case 1), the guarantee that the application only sees complete images (Case 2), and that the application sees no data loss (Case 3). The base network configuration corresponds to Platform 7 in Table 2, with two changes introduced 25 seconds and 50 seconds into the experiment. The first change degraded the bandwidth between the client and the gateway ($L2$) to 440 Kbps from the original 1 Mbps. The second change modeled the transition of the network from (secure) wired connectivity to (insecure) wireless connectivity. Since our focus was on measuring the overheads of the reconfiguration procedure, our experiment had an external procedure generate the necessary events and coordinate with the "sandbox" code to control bandwidth available to the application.



Gateway N_1
Client N_2
F
F R
F R E
D
A
B
C
F ImageFilter
R ImageResizer
E Encryption
D Decryption
Start of Reconfiguration
End of Reconfiguration (Case 1 and 2)
End of Reconfiguration (Case 3)
A
B
Start of Reconfiguration
End of Reconfiguration (Case 1 and 2)
End of Reconfiguration (Case 3)
C
Time
0.72
1.45
1.10
1.66
compute new plan
diff plans
RMI to upstream and downstream
RMI from downstream (ready)
RMI (ack)
0.371
0.005 convey plan to delegate
0.092
0.228
0.054
generate next semantic segment
0.172 wait for downstream to get segment
0.260
0.169
0.099
all done

Figure 9: Path reconfiguration in the image streaming application. All times are in seconds.

Figure 9 shows the paths created by the planning procedure in response to the events (top left), and how the reconfiguration procedure transitions among these paths along the execution timeline (top right). The initial path **A** contains only an ImageFilter component running on the gateway node. The first event, triggered when bandwidth drops, results in the introduction of an additional component, the ImageResizer, on the gateway node (path **B**). Note that CANS reconfiguration is accomplished completely automatically, without any involvement from the application code. Depending on the semantics that need to be supported, the total time for reconfiguration is either 0.72 seconds (for Cases 1 and 2) or 1.45 seconds (for Case 3). The path again gets reconfigured when the second event is received, corresponding to the switch between wired and wireless connectivity; the new path **C** now contains Encrypter and Decrypter components on the gateway and client nodes to ensure secure transmission. As before, reconfiguration is achieved automatically, but incurs slightly larger overheads (1.10 seconds for Cases 1 and 2, and 1.66 seconds for Case 3), because of the additional work involved in orchestrating reconfiguration activities across multiple nodes.

Reconfiguration overheads of 1-2 seconds are acceptable for most applications running in mobile access scenarios. However, to better understand the contributing factors, we broke down the 1.45 seconds required for Case 3 reconfiguration into four stages (bottom part of Figure 9): (1) construction of a new plan and computing the delta from the current plan; (2) RMI calls to the upstream and downstream points to start buffering and monitoring; (3) waiting for the reconfiguration condition to become true; and (4) the two-phase procedure to install the new path and resume data transmission. Note that stage 3 takes different times for different reconfiguration semantics (actually for this application, stage 3 can be bypassed for both Case 1 and Case 2), and stage 4 can be combined with stage 2 when it is possible to infer that the reconfiguration condition is immediately satisfied (always true for Case 1). The cost breakdown shows that the dominant contributors are plan creation (0.48 seconds) and cross-node handshaking using RMI (0.66 seconds), with application-independent steps (gray blocks) incurring negligible overhead. These results are encouraging because our current work has the potential to substantially reduce these two overheads: the first because of a modified planning procedure, which reuses previously computed partial solutions, and the second by avoiding use of RMI, instead leveraging an efficient control channel between execution environments.

7 Related Work and Discussion

The research described in this paper is very closely related to several recently proposed infrastructures that aim to augment the traditional notion of a network path with injected application-specific components, either only at the end points [14, 8, 11] or throughout the path [17, 10, 2, 20, 4, 1, 22, 6]. Rather than describe all such systems, we focus our attention here on the subset which offer some form of automatic support for path creation and reconfiguration.

The Ninja project's Automatic Path Creation (APC) service [6], also used in the Universal Inbox infrastructure [15], can be used to create paths between various end devices and services. Both APC and our approach formulate the component selection problem in terms of type compatibility, however, there are significant differences. At a high level, unlike the performance-oriented focus of our work, APC is a function-oriented method, which ignores network link properties and node and link resource constraints. A consequence of this difference is that a shortest-path approach to planning suffices for Ninja (with the restriction that a data type can appear only once along a path), while we need a more sophisticated dynamic programming-based approach. Other differences include our support for path reconfiguration and a more general notion of data, stream, and augmented types, which were motivated by a desire to model link characteristics in a unified fashion and contrast with Ninja's notion of a relatively simple string type.[2]

Kiciman and Fox [9] have proposed a general path infrastructure framework for composing mediators distributed across a network of machines. This infrastructure builds upon Ninja's APC service and suffers from the same limitations. Furthermore, this approach separates out logical path creation (choice of components) from the mapping of components to physical resources. As we have shown in Section 6, such decoupling can produce suboptimal solutions because of poor or redundant component placement.

Recent work in the Scout project [12] has looked at a template based path construction algorithm for delivering media objects that takes into consideration the latter's resource requirements, user preferences, node capabilities, and programmer-provided path rules. This work shares its performance focus with ours, however, the primary difference arises from the fact that unlike our high-level type-driven approach, here a programmer must a priori construct path templates and store them into a central database. The Scout algorithm takes a lower-level approach, simply choosing an appropriate template and instantiating it based

[2] We must note that Ninja's researchers believe that types are inadequate as a lookup mechanism for components since they do a poor job of capturing semantics. To an extent, this concern is alleviated by our extended notion of data types that can include type-specific attributes. More importantly, the component selection problem can be viewed at multiple levels: semantics-driven selection figures at the highest level, whereas type-based selection and mapping, which are the focus of this paper, are lower-level but equally important concerns.

on other programmer-provided rules that decide whether or not a component can be created on a resource. We avoid this last problem because of the application-level nature of our components, which rely on a relatively standard execution environment interface (the Java virtual machine in our case). On the flip side, the Scout approach does a better job of modeling low-level resource properties such as the availability of a specific kind of video hardware or NIC.

The Panda project [16] also proposes a planning scheme for optimally placing network-level components to modify an application's data stream in response to unfavorable network conditions. While two schemes are discussed, one based upon selection from a reusable plan set and the other based on exhaustive constraint space-based search, to the best of our knowledge these schemes have not yet been implemented or evaluated with real applications.

Our work is also complementary to emerging standards for delivery of content to small devices with different user preferences in that it aims to automate the process of setting up these delivery paths. Two such standards have been proposed recently: the CC/PP (Composite Capabilities/Preference Profiles) protocol from the World Wide Web Consortium (W3C) [21], and the UserAgent protocol from the Wireless Application Protocol (WAP) forum [19].

To the best of our knowledge, the approach described in this paper is one of the first schemes to not only consider the functionality of the data path, but also takes both network link properties and node resource constraints into account. Our work is also one of the first to perform a detailed evaluation of the overhead of path creation, and reconfiguration, and measure the performance of the deployed paths. While we expect the performance to improve as the CANS implementation is further tuned, the numbers in this paper provide a concrete baseline for the potential of automatic approaches for constructing network-aware access paths.

8 Conclusions

This paper has presented an automatic approach for the dynamic deployment of intermediary components along client-server paths, which can be efficiently reconfigured at run time, to enable ubiquitous, network-aware access to internet services. This approach leverages a type-compatibility formulation of the problem, which takes as input only high level specifications of component behavior and network route characteristics. Novel to this formulation is the fact that constraints due to node and network link characteristics are naturally integrated into the type model simply by modeling the latter as entities that transform the type of data passing across them. This formulation lends itself to a dynamic programming based polynomial-time algorithm, which simultaneously selects and maps appropriate components to optimize a global metric such as client throughput or response time. This algorithm is complemented by an efficient semantics-preserving data path reconfiguration strategy. Experiments with the planning algorithm and reconfiguration in the contexts of a web access scenario and an image streaming application using the CANS infrastructure under various network and end device characteristics have verified that automatic path creation and reconfiguration is both feasible and can yield substantial performance benefits. Thus, in contrast to current-day static access paths to internet services, our work argues for a flexible approach where paths leading to these services are automatically and dynamically composed to satisfy user preferences and network resource constraints.

CANS is one component of a larger project, Computing Communities, which focuses on distribution middleware for legacy applications. Our future work involves generalizing the CANS planning algorithms to handle efficient reconfiguration in the context of multi-ported components, and integrating CANS with related efforts emphasizing resource management and security issues.

Acknowledgements

The authors thank Zvi Kedem, Anatoly Akkerman, and Tatiana Kichkaylo for their contributions to various parts of the CANS infrastructure and its algorithms. This research was sponsored by DARPA agreements F30602-99-1-0157 and N66001-00-1-8920; by NSF grants CAREER:CCR-9876128 and CCR-9988176; and Microsoft. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies or endorsements, either expressed or implied, of DARPA, Rome Labs, SPAWAR SYSCEN, or the U.S. Government.

References

[1] E. Amir, S. McCanne, and R. Katz. An Active Service Framework and its Application to Real-time Multimedia Transcoding. In *Proc. of the SIGCOMM'98*, August 1998.

[2] A. T. Campbell and et al. A Survey of Programmable Networks. *ACM SIGCOMM Computer Communication Review*, April 1999.

[3] F. Chang, A. Itzkovitz, and V. Karamcheti. User-level Resource-Constrained Sandboxing. In *Proc. of the 4th USENIX Windows Systems Symposium*, August 2000.

[4] A. Fox, S. Gribble, Y. Chawathe, and E. A. Brewer. Adapting to Network and Client Variation Using Infrastructural Proxies: Lessons and Prespectives. *IEEE Personal Communication*, August 1998.

[5] X. Fu, W. Shi, A. Akkerman, and V. Karamcheti. CANS:Composable, Adaptive Network Services Infrastructure. In *Proc. of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS)*, March 2001.

[6] S. D. Gribble and et al. The Ninja Architecture for Robust Internet-Scale Systems and Services. *Special Issue of IEEE Computer Networks on Pervasive Computing*, 2000.

[7] J. Haartsen. BLUETOOTH– The universal radio interface for ad hoc, wireless connectivitity. *Ericsson Review*, 1998.

[8] A. D. Joseph, J. A. Tauber, and M. F. Kasshoek. Mobile Computing with the Rover Toolkit. *IEEE Transaction on Computers:Special Issue on Mobile Computing*, 46(3), March 1997.

[9] E. Kiciman and A. Fox. Using Dynamic Mediation to Intergrate COTS Entities in a Ubiquitous Computing Environment. In *Proc. of the 2nd Handheld and Ubiquitous Computing Conference (HUC'00)*, March 2000.

[10] A. Mallet, J. Chung, and J. Smith. Operating System Support for Protocol Boosters. In *Proc. of HIPPARCH Workshop*, June 1997.

[11] R. Mohan, J. R. Simth, and C.S. Li. Adapting Multimedia Internet Content for Universal Access. *IEEE Transactions on Multimedia*, 1(1):104–114, March 1999.

[12] A. Nakao, L. Peterson, and A. Bavier. Constructing End-to-End Paths for Playing Media Objects. In *Proc. of the OpenArch'2001*, March 2001.

[13] B. Noble. System Support for Mobile, Adaptive Applications. *IEEE Personal Communications*, pages 44–49, February 2000.

[14] Brian D. Noble. *Mobile Data Access*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1998.

[15] B. Raman, R.H. Katz, and A. d. Joseph. Universal Inbox: Providing Extensible Personal Mobility and Service Mobility in an Integrated Communication Network. In *Proc. of the Workshop on Mobile Computing Systems and Applications (WMSCA'00)*, December 2000.

[16] P. Reiher, R. Guy, M. Yavis, and A. Rudenko. Automated Planning for Open Architectures. In *Proc. of OpenArch'2000*, March 2000.

[17] P. Sudame and B. Badrinath. Transformer Tunnels: A Framework for Providing Route-Specific Adaptations. In *Proc. of the USENIX Technical Conf.*, June 1998.

[18] U. Varshney and R. Vetter. Emerging Mobile and Wireless Networks. *Communications of the ACM*, pages 73–81, June 2000.

[19] WAP. WAP Forum Specifications. Technical report, http://www.wapforum.org/what/technical.htm, January 2000.

[20] D. J. Wethrall, J. V. Guttag, and D. L. Tennenhouse. ANTS: A toolkit for building and dynamically deploying network protocols. In *Proc. of 2nd IEEE OPENARCH*, 1998.

[21] W3C CC/PP Workgroup. CC/PP specification. Technical report, http://www.w3c.org, August 1999.

[22] M. Yavis, A. Wang, A. Rudenko, P. Reiher, and G. J. Popek. Conductor: Distributed Adaptation for complex Networks. In *Proc. of the Seventh Workshop on Hot Topics in Operating Systems*, March 1999.